Separating Essentials from Incidentals: An Execution Architecture for Real-Time Control Systems

Daniel Dvorak and Kirk Reinholtz
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
818-393-1986
{daniel.dvorak,william.k.reinholtz}@jpl.nasa.gov

Abstract

Large multi-threaded real-time control systems can be notoriously hard to verify because source code often intertwines several concerns such as functionality, control flow, data flow, synchronization, timing, protocol, and runtime efficiency. Knowing that these concerns can interact in unforeseen ways, system-level testing generates only minimal confidence in a system's dependability. This paper describes an execution architecture that makes such systems far more analyzable and verifiable by aggressive separation of concerns. The architecture separates two key software concerns: transformations of global state, as defined in pure functions; and sequencing/timing of transformations, as performed by an engine that enforces four prime invariants. The important advantage of this architecture, besides facilitating verification, is that it encourages formal specification of systems in a vocabulary that brings systems engineering closer to software engineering.

1. Introduction

1.1 Sequencing

The Jet Propulsion Laboratory (JPL)—part of the California Institute of Technology—serves as NASA's lead center for the robotic exploration of space. For many years, JPL's spacecraft have employed a top-level execution mechanism known as "sequencing" in which parameterized commands are sent to tasks at specific instants in time. Typically, a sequence engine runs multiple sequences concurrently. Sequences are designed by ground controllers, carefully checked and simulated, and then transmitted to the spacecraft where they are executed at the specified time. The end result is that the timing and ordering of activities onboard is highly predictable. That predictability has long been a comfort to mission operations, but it has become increasingly problematic as JPL conducts more *in situ* mission of exploration, such as running rovers on the surface of Mars.

1.2 MDS Architecture

Given the long round-trip light-time delays of planetary missions and the need for rovers and other spacecraft to react more quickly to events and hazards, JPL has invested in a significant new architecture in which top-level control is based on goals rather than commands and in which execution timing is based on goal success/failure as well as temporal constraints. This architecture, named MDS (Mission Data System), is based on the unifying notion of physical *state* and how a control system estimates and controls state [1]. MDS emphasizes explicit representation of physical states (both continuous and discrete states), explicit models of hardware and physical effects, and goal-oriented operation that enables varying levels of onboard autonomy.

The MDS architecture cleanly separates several concerns: it separates estimation from control, it separates models of how things work from decision-making, it distinguishes between evidence and state knowledge, it separates deliberative and reactive control, it distinguishes between goals and commands, and it separates data management from data transport.

As shown in Figure 1, real-time control loops in MDS involve four kinds of components: hardware adapters, state variables, estimators, and controllers. There is a hardware adapter for each controllable hardware unit, and each one provides software interfaces for sending commands and obtaining measurements. A state variable is a component that holds information about a physical state (such as rover position) and whose value history is made available as telemetry. An estimator interprets measurements from potentially multiple sensors in order to generate state estimates. A controller compares current state estimates to a 'goal' (a constraint on the value of a state variable over a time interval) and issues commands to actuators, as needed, to influence a physical state.

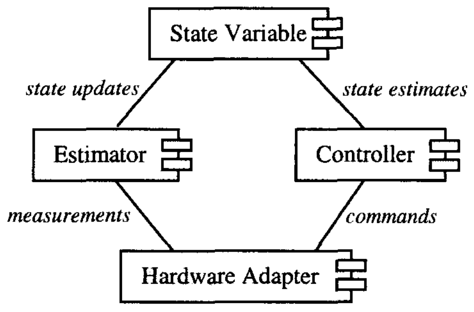

Figure 1. A simple hard real-time control loop in MDS involves data flows among four components.

The dominant data flow around a control loop involves four flows: controllers query state variables for state estimates; controllers submit commands to hardware adapters; estimators query hardware adapters for measurements; and estimators update state variables.

The MDS architecture focuses on the things that a systems engineer must analyze and specify, but is less explicit about a number of things that must be decided in a practical real-time software implementation. This paper presents an expansion of the MDS architecture to address issues of *execution architecture*.

2. Proposed Execution Architecture

2.1 Pure Functions, Engine, and Prime Invariants

The proposed architecture, called "Verifiable MDS Architecture" (VMA), separates two key software concerns. First, each individual transformation of global state is written as a pure function: one whose results depend *only* upon its arguments. Second, the sequence of transformations is managed by an engine that must maintain certain invariants, but is otherwise free of design constraint (this is a pure abstraction — if it acts like an "engine", it's an engine).

All engine implementations are required to maintain the following invariants. The purpose of these invariants is to ensure that the engine is aware of all inputs and outputs of all functions (so it can manage interactions and dependencies), and that the functions have transaction semantics so that interactions can be completely managed by the engine without knowledge of the semantics of the functions. Transformation semantics are thus strictly determined by the function code, and interactions are strictly determined by the engine. The concerns are separated. The following are the engine's *prime invariants*:

1. No function will ever see any of its arguments change in value while that function is executing (though they can be allowed to change while the function is executing, the function won't see those changes: it in effect operates upon captured copies of the values).
2. Each function is annotated to use either:
    a. *value semantics*, in which case the arguments are in effect copied for use by the function (and so they can be written by others while the function is executing); or
    b. *reference semantics*, in which case the engine ensures that the arguments are neither read nor written by other functions while the function is executing.
3. For all functions that return multiple values, those values will be updated simultaneously insofar as any function can detect (this greatly reduces some branching in correctness analysis).
4. No function will make reference to or update any state whatsoever other than that passed directly as arguments to that function.

Note that the invariants above are not sufficient to determine the order in which the functions must be executed. This is intentional, because we want to allow an imperative "virtual machine" form of engine where the order is directly specified, rather than deduced. That said, later work may attempt to formalize the dependency information such that it can become part of the invariants. Time will tell. For now we want to enable at least one "obvious" engine implementation.

2.2 Benefits

1. Separation of concerns: A driving advantage is that this approach separates the management of interactions (protocol) from the semantics of a particular transformation. Many of the concrete advantages in this list come of this separation.

2. Semantics do not depend on the specifics of the engine. As long as the engine obeys the prime invariants, the system will work. It is thus reasonable to have different engines for different needs, e.g. one that has distributed computation and another that doesn't.

3. Encourages formal specification of the system. The approach adds a couple of formal elements to the system concept: transformation functions and interactions. These items should enter the vocabulary of the systems engineers, giving them one more direct connection between systems engineering and the code.

4. Because of the additional formality in modeling and interactions, this should make the architecture an enticing target for and benefactor of future work in formal modeling, specification, and analysis.

5. The MDS concept of models, such as state effects models and measurement models and command models, are naturally expressed as pure functions.

6. Since the functions are pure by definition, then they are:

    a. easier to write correctly than functions that contain internal state;

    b. easier to unit test due to limited reference to the environment;

    c. easier to write and runtime confirm pre- and post-conditions, perhaps in the form of pre_f(...) and post_f(...) for each f(...) that can be executed by the engine, or completely elided during production; and

    d. exception safety is trivial. If the function throws an exception, it is simply aborted without updating the system state. Since it's a pure function, it can't leave the system in an intermediate state. Of course recovery is still an issue, because the function will throw again if given the same arguments.

7. Reversible system. If you save each version of each output of each function, and keep track of the order in which things occurred, you could actually reverse execution of the system. That could aid in fault recovery, and could make debugging easier because the whole history of the evolution of the system is there to examine.
8. Easier to design and implement the system because the systems engineers probably specify most of the functions and their rules of interaction; programmers can focus on the function bodies.
9. Can abort a function during execution because it's pure. This gives another option in handling real-time overruns.
10. Easy to execute in parallel/distributed manner. A consequence of separating function semantics from interactions is that functions that don't interact can be executed in parallel. Most of the engine types described below trivially utilize multiple threads, but the function bodies are completely independent of that: there are no locks or other synchronization mechanisms anywhere in those functions.
11. Scales to parallel and distributed systems. The approach scales more easily than e.g. components, because the essentials of the interactions are known by the system.
12. Minimal use of the heap. Since the approach puts the significant data structures into global memory, and function computations are on the stack, and the basic computation model is oriented towards discrete transformations, it may turn out that the heap is used less than in a component architecture. Use of the heap greatly increases the difficulty of proving a system correct, so reduced heap use should translate to lower development costs.
13. Suitable for RTSJ implementation. VMA encourages the use of only global memory ("immortal memory") for the state and the stack (stack + "scoped memory") for the transformation functions. It is very likely that this will make it straightforward to render VMA into RTSJ, and unlikely to be full of corner cases.
14. Events and notifications are elevated into the architecture.
15. Debugging - System can be single-stepped, run backwards, stopped...

2.3 The Engine

There are numerous styles of engine that could execute the proposed architecture, and the architecture is defined to allow this. This section develops a complete specification for a "classical" sensor→actuator loop in MDS, using a virtual machine style of engine. The following code fragment captures the basic MDS control loop. The comments draw attention to some key points.

```
-- Example of MDS control loop in VMA.
-- NOTES
--  Notation
--    SV* a state variable
--    VH* a value history
--    HAread a hardware adaptor that reads a sensor
--    HAwrite is a hardware adaptor that writes to actuator.
--    m a measurement
--    dm a distilled measurement
--    Anything on LHS is in global state


-- For now just assume this executes top to bottom,
-- as part of a rate group.
-- It's important you remember the execution model:
--   1. Each function is allowed to reference ONLY the state passed as arguments.
--   2. It's permitted but not encouraged to create global objects of ad-hoc state,
--      called CS* here, to preserve state across function calls.
--   3. The arguments are constant and are in effect copies of the state.
--      a. If the state is too large to allow value semantics, a function
--         can be annotated that it used reference semantics.  In that case
--         the args are blocked for both read and write by other functions,
--         for the duration of execution of the function.  Abort is not
--         supported in this case.
--   4. The return values are written atomically back into the global state.


-- Run everything between the curly braces at 5Hz.  If there are
-- lots of these in the 5Hz rate group, you can either write
-- them as separate cyclic groups like this, in which case
-- their interactions are not managed, or you can put them
-- sequentially in a single cyclic to force a particular
-- order.  In real life cyclic() will have deadline
-- mechanisms, not shown here.  Thanks to the abortable
-- nature of pure functions, we can also cleanly abort any
-- operations that overrun their time bound, again not shown.

cyclic(5Hz){

  -- This set reads a measurement, distills it, and updates
  -- the value history of the state variable.  You don't
  -- have to use the distillation idiom, of course.

  VHm   = HAread(VHm)                  -- New measurements(s) appended to VHm
  VHdm1 = est1(VH, SVother...)         -- distill it and append to VHdm
  VHdm2 = est2(VHdm2, VHdm1, SVother...) -- distill more if you wish
                                       -- Can distill zero or more times
  SV.vh = est(SV.vh, VHdm2, SV.xgoal)  -- Final estimate.  Note only vh updated.


  -- The control process uses the value history and x-goal to
  -- compute a command, if any.  It is a principle of the
  -- state architecture that there is only zero or one
  -- command "between" the controller and the hardware
  -- adapter at any time, that's why there is no command
  -- queue.  The engine must run the ctrl() and HAwr() in a
  -- coordinated manner so the cmd doesn't get overwritten.

  VHstatus,cmd = ctrl(SV.vh,SV.xgoal,VHstatus) -- Compute new command if any.
  VHstatus,cmd,VHc = HAwr(cmd,VHc,VHstatus)    -- Do stuff + move cmd into VHc and
                                               -- empty cmd slot.
                                  -- Now the cmd is in the VHc, indicating that the
                                  -- hardware is aware of the command and so should
                                  -- be considered in the estimation process.
                                  -- ctrl() and HAwr() should return immediately, not
                                  -- loop waiting for the xgoal to either succeed or
                                  -- fail.
}
```

2.3.1 *Events*

The engine as described so far can only execute sequences of functions in a cyclic manner. However, the MDS concept includes event-triggered execution. For example, when an executable goal (xgoal) succeeds, fails, or is aborted, that event initiates computations that result in (among other things) a new xgoal for the goal achiever. Although this could be implemented using polling, it is more naturally an event, and so this architecture should support events.

Any change of state can raise an event. A predicate written on the state plus changes to state can be used to guard a sequence of functions. The engine will cause that sequence to execute some time after the predicate becomes true. In effect, every time any state is changed, all predicates are tested, and any that are true are marked as true. The actual implementation should be more efficient than that, but the effect is identical.

This code fragment shows how to handle a change to goal status.

```
-- Example of event-triggered execution.

  -- These are from the first example.  Each operation that might
  -- note a change to status simply updates a state item "status".

  VHstatus,cmd = ctrl(SV.vh, VHstatus, SV.xgoal)
  VHstatus,cmd,VHc = HAwr(cmd, VHstatus, VHc)

  -- The predicate in this "when" sequencing block is, in effect,
  -- checked every single time anything changes in the state of the
  -- system.  If the predicate is true, the block is marked for
  -- eventual execution.  We'll discuss the timing of this in the
  -- "real-time" section that follows.

  when(latest(VHstatus) == goalfailed){
    -- whatever ...
  }
```

2.3.2 *Exogenous events*

It may be useful to allow code outside of our world (e.g. from a device driver) to interact with code inside of our world (e.g. insert a new status value into VHstatus) in a "push" mode where the outside entity controls when the interaction occurs. We can already do "pull" style, via hardware adaptors or some less formal variant thereof (i.e. a "hardware adaptor" that reads performance counters instead of a gyro.

Consider goal failure, for example. A goal might fail during the execution of the achiever, when the achiever discovers it can't find any command that will keep the state trajectory within the constraints. Or, the goal might fail when a device driver interrupt handler notices some register value has gone out of bounds. In the latter case, the most timely response occurs when the driver itself can indicate that something has happened (e.g. goal status goes from OK to failed), so the engine notices it ASAP.

Pushes from the outside world have great potential to complicate the compute model and make it less robust. For now we prefer a very restricted model, something like this, which maps the external event into normal operations as far upstream as I can come up with:

1. The external code "pushes" notification of its interest in being noticed via a semaphore.
2. The engine, at some well defined point in its execution, checks the semaphore.
3. If the engine finds that a notification has occurred, an insertion into the value history bound to that notification is executed (perhaps a timestamp is inserted, or an incrementing counter).
4. From that point on, the system behaves exactly as though that insertion occurred through a normal operation on a VH.

Note this interface is absolutely *not* available to normal code! Normal code can update stuff in the normal way, and it is *very* important that it remains this way. Otherwise this breaks the "pure function" model, the interactions become hidden, and other bad things happen. As a reminder, here's the normal way to track status:

```
-- Normal way to maintain status information on some operation
...,VHstatus,... = func(...,VHstatus,...)
...
when(latest(VHstatus) == Failed){ ... }
```

2.3.3 *Parallel execution*

In this model of the engine, some of the dependencies are captured only in the sequencing of the function calls. The engine will not perform analysis beyond that because we haven't had time to define how to do so and because the MDS state architecture is supposed to be resilient in the face of such conditions. But, for the record, here's the kinds of races that could occur:

```
b = f(a)
c = g(b)   -- Dependency: Can't start until f(a) completes.
           -- Also called RAW: Read After Write hazard.

b = f(a)
a = g(c)   -- Antidependency: Can't start until f(a) completes.
           -- Also called WAR: Write After Read hazard.

b = f(a)
b = f(c)   -- Write ordering: Must write second b second.
           -- Also called WAW: Write after Write hazard.
```

Within a single sequence of operations—say the body of a cyclic(){...}—the engine can cheaply detect the above conditions, and execute the body in parallel with synchronization to improve performance. This is what super-scalar processors do, and approaches for scheduling in the face of these hazards are well defined (e.g. see Hennessey and Patterson). It would be easy to implement the engine so that it uses multiple CPU threads to speed up a program like the above using super-scalar concepts, without violating any of the causal types noted above.

3. Summary

The architecture described above is simple in its separation of transformations and the applications of transformations, and it is formal in a way that invites formal analysis for verification. Pure functions are easy to write and easy to test, and they never have to use synchronization mechanisms (a common source of defects in multi-threaded programs). This makes them far more reusable than code that tries to handle concerns other than pure transformations. Importantly, the architecture elevates "protocol" into the architecture; the ordering and data dependencies among transformations become explicit inputs to the engine. Similarly, timeliness characteristics such as periods, costs, and deadlines are inputs to the engine and are *not* embedded in code for transformations.

4. Acknowledgements

This work was performed at the Jet Propulsion Laboratory of California Institute of Technology under contract with the National Aeronautics and Space Administration. We thank the Office of the Chief Technologist for funding under the

Research & Technology Development (R&TD) Program, and the strong support of the R&TD committee on Advanced Software Techniques & Methods Initiative.

5. REFERENCES

[1] Dvorak, D., Rasmussen, R., Reeves, G., and Sacks, A. Software Architecture Themes in JPL's Mission Data System. Proceedings of the 2000 IEEE Aerospace Conference, Big Sky, Montana, March, 2000.

[2] Hennessy and Patterson, TBD.